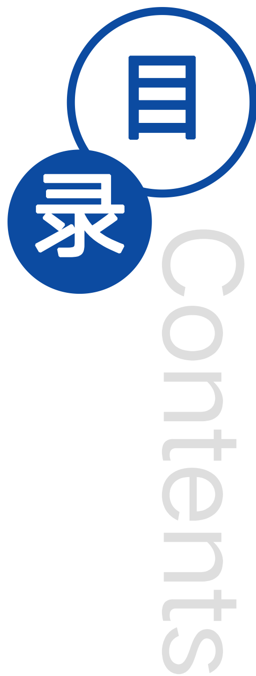
隐私数据加密最佳实践

熊高祥、何鹰

2022年 09日

目录



1. 背景
2. 业内通行技术方案
3. 实现方案
4. 集成使用示例

背景

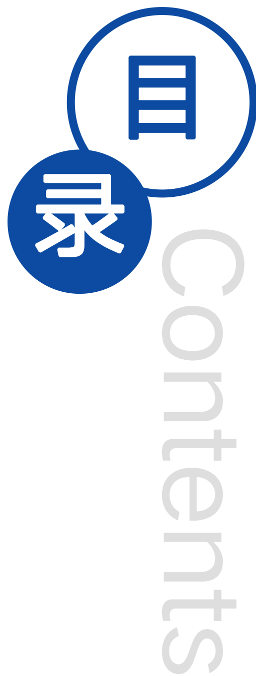
随着数据泄露事件在大数据时代层出不穷，随着行业新网络形态、新技术以及新应用场景的发展，新的数据类型、数据生产方式、数据处理方式和终端形式不断涌现，数据安全挑战也随之加剧。国家已出台各级各行业的法律法规及配套文件，不断加大数据安全与隐私保护的监管力度。

根据国家数据安全发和个人信息保护法要求，系统中的个人信息字段必须做加密传输和存储。根据网络安全法的要求：“传输和存储个人敏感信息时，应采用加密等安全措施”是强制要求。

数据安全隐私相关的法律文件

时间	法律文件
2021年11月1日	全国人大会常务委员会发布**《个人信息保护法》**
2021年6月10日	全国人大会常务委员会发布**《数据安全法》**
2020年3月6日	信息安全标准化技术委员会发布**《个人信息安全规范》**
2019年5月28日	国家互联网信息办公室发布**《数据安全管理办法》**
2016年11月7日	全国人大会常务委员会发布**《网络安全法》**

目录



1. 背景

2. 业内通行技术方案

3. 实现方案

4. 集成使用示例

业内通行技术方案—数据加密

数据加密简介

数据加密方案有许多，主流的有磁盘加密、TDE加密、应用层加密。

- 磁盘加密
  - 磁盘加密技术通过对磁盘进行加密以保障其内部数据的安全性。
  - 等保三级不一定可以过
- TDE加密
  - 由数据库厂商在数据库引擎中实现，数据在写入磁盘之前进行加密，从磁盘读入内存时进行解密
  - Mysql的TDE免费、PG的TDE收费、Oracle的TDE单独收费
  - 可以通过等保三级
- 应用层加密
  - 在应用系统层的源代码中对敏感数据进行加密，加密后将密文存储到数据库中。可以直接在应用系统或者中间件中以独立的函数或模块形式完成加密。
  - 严格来说不再是针对数据库或者文件加密，而是针对数据进行加密。

场景	磁盘加密	TDE	应用层加密
物理磁盘盗走	安全	安全	安全
服务器账号泄露	不安全	安全	安全
数据库账号泄露	不安全	不安全	安全
加密算法和秘钥泄露	不安全	不安全	不安全

业内通行技术方案—应用层数据加密

应用层加密怎么做

数据库明文存储，诸如内部高权用户或侵入数据库服务器的黑客，都可以毫无阻碍地访问数据库的重要数据，一些敏感数据在存储期间的机密性并不能得到有效的保障。

根据信息安全的目标，需要对应用生产数据加密，达到满足安全要求，预防存储层明文泄密，实现自主可控安全。

- 原字段加密存储
  - 选择符合安全的加密算法，推荐优先使用国密算法

  对称：SM4，非对称：SM2），国际算法（对称：AES，非对称：RSA）
  - 敏感数据在插入和修改的时候需要按照算法加密后写入数据库
  - 敏感数据查询出来之后需要解密显示

  缺点：

  该方案可以解决数据库存储原文的问题，但是由于加密算法是将整串字符串加密，导致模糊搜索无法实现。

业内通行技术方案—加密后模糊查询

加密后的数据如何进行模糊查询

- 加载数据到内存数据库
  - 将所有数据加载到内存数据库中进行解密，这样我们就和查询原文一样了。这个方案虽然可以实现模糊查询，如果数据量小的话可以使用这个方式来做，这样做既简单又实惠，如果数据量大的话那就是灾难。
- 数据库中实现与程序一致的加解密函数
  - 修改模糊查询条件，使用数据库解密函数先解密后再模糊查找，这样做的优点是实现成本低，开发使用成本低，只需要将以往的模糊查找稍微修改一下就可以实现。但是数据库中密文和加密函数存储在一起，无法应对数据库账号泄漏场景。

```
原生SQL: select * from user where name like "%xxx%"
实现解密函数之后: select * from user where decode(name) like "%xxx%"
```

- 脱敏存储
  - 对密文数据进行脱敏后存储到模糊查询列，该方案会丢失太多精度。

例如：手机号为13012345678通过AES加密得到密文列phone，通过脱敏算法得到phone_mask列

id	name	sex	phone	phone_mask
1	qyPX2iE4jasGZIUCJvP8J	女	V0H95UPxrPKdTj4D5Zek+g==	130****5678

业内通行技术方案—加密后模糊查询

分词组合加密存储

对密文数据进行分词组合，将分词组合的结果集分别进行加密。对字符进行固定长度的分组，将一个字段拆分为多个，比如说根据4位英文字符，2个中文字符为一个检索条件，举个例子：

ningyu1使用4个字符为一组的加密方式，第一组ning，第二组ingy，第三组ngyu，第四组gyu1 … 依次类推，全部加密之后存储到模糊查询列。

如果需要检索所有包含检索条件4个字符的数据比如：ingy，加密字符后通过 key like “%partial%” 查库。

淘宝、拼多多、JD他们的API，他们对平台订单数据中的用户敏感数据就是加密的同时支持模糊查询，使用就是这个方法。(url加上)

缺点：

- 存储成本增加
  - 自由分组会增加数据量
  - 加密后长度会增长
- 模糊查询长度有限制
  - 由于安全问题，自由组合长度不能太低，不然可以采取彩虹表破解。
  - 对模糊查询的字符长度是有要求的，以我上面举的例子模糊查询字符原文长度必须大于等于4个英文/数字，或者2个汉字才能搜索到结果

https://open.taobao.com/docV3.htm?docId=106213&docType=1
https://jaq-doc.alibaba.com/docs/doc.htm?treeId=1&articleId=106213&docType=1
https://open.pinduoduo.com/application/document/browse?idStr=3407B605226E77F2
https://jos.jd.com/commondoc?listId=345

原创技术方案—单字符摘要算法

单字符摘要算法

上述方案都能使用，但是有没有更好的方案。我们发现单字符加密存储是性能和搜索都能兼顾的，但是不符合安全要求，那怎么办。受到脱敏算法和密码散列函数的启发，我们发现丢失数据、和单向函数这个思路我们可以借鉴。

- **密码散列函数应该有的四个主要特性**
  - 对于任何一个给定的消息，它都很容易就能运算出散列数值。
  - 难以由一个已知的散列数值，去推算出原始的消息。
  - 在不更动散列数值的前提下，修改消息内容是不可行的。
  - 对于两个不同的消息，只有极低的几率会产生相同的散列数值。

安全性：因为有单向函数，所以是不可以推算原始消息的。因为我们想模糊搜索精确度提升，所以想单字符加密，但是又会被彩虹表破解。

所以我们借鉴单向函数（保证每个字符加密之后的字符一致），然后增加碰撞（保证每个字符串逆向结果为1：N），这样就大大加强了安全性。

原创技术方案—单字符摘要算法

```
// ascii 偏移量 防止出现原文
private int delta = 1;

// 二进制mask用于丢失精度
private int mask = 0b0111_1111_1101;

//3726个常用汉字
public static String DICT = "啊阿埃挨。。。。";
public String encrypt(final String plainValue,
final EncryptContext encryptContext) {
        if (null == plainValue) {
            return null;
        }
StringBuilder sb = new StringBuilder(plainValue.length());
        plainValue.chars().forEachOrdered(c -> {
            int masked = (c + delta) & mask;
            // 兼容模糊查询 % 不处理
            if ('%' == c || '%' == masked) {
                sb.append((char) c);
            } else {
                if (c > 256) { // 非数字字母(包括拉丁) 取常用汉字
                    masked = DICT.charAt(masked);
                }
                sb.append((char) masked);
            }
        });

        return sb.toString();
    }
```

- 先定义二进制mask码用来丢失精度0b0111_1111_1101
- 所以我们采取单个字符串的ASCII码进行加1防止任何原文出现在数据库。
- 然后将字符串ASCII码转换成二进制和mask进行与运算，把原字符串偏移之后的二进制码的2位的数字丢失，保留11位。
- 数字字母（包括拉丁文都）丢失精度之后直接输出
- 汉字丢失精度后转成十进作为index去常用字中取字

原创技术方案—单字符摘要算法

单字符摘要算法解释：

第一版算法：

Mask：0b1111111111001111101

原字符：0b1000101110101111 讯

加密后：0b1000101000101101 設

假设我们知道秘钥和加密算法的情况下，反推出来的原字符串为

1. 0b1000101100101101 謭
2. 0b1000101100101111 謯
3. 0b1000101110101101 训
4. 0b1000101110101111 讯
5. 0b1000101010101101 読
6. 0b1000101010101111 誯
7. 0b1000101000101111 訯
8. 0b1000101000101101 設

这样每个字符串会根据丢失的位数，反推出来的结果为2^n。因为常见汉字的unicode码在十进制时间隔都很大，可以发现反推出来的汉字基本都不是常用字，反推出原字的几率比较大。

啊:21834 阿:38463 埃:22467 挨:25384 哎:21710 唉:21769 哀:21696
皑:30353 癌:30284 蔼:34108 矮:30702 艾:33406 碍:30861 爱:29233
隘:38552 鞍:38797 氨:27688 安:23433 俺:20474 按:25353 暗:26263
岸:23736 胺:33018 案:26696 肮:32942 昂:26114 盎:30414 凹:20985
敖:25942 熬:29100 翱:32753 袄:34948 傲:20658 奥:22885 懊:25034
澳:28595

芭:33453 捌:25420 扒:25170 叭:21485 吧:21543 笆:31494 八:20843
疤:30116 巴:24052 拔:25300 跋:36299 靶:38774 把:25226 耙:32793
坝:22365 霸:38712 罢:32610 爸:29240 白:30333 柏:26575 百:30334
摆:25670 佰:20336 败:36133 拜:25308 稗:31255 斑:26001 班:29677
搬:25644 扳:25203 般:33324 颁:39041 板:26495 版:29256 扮:25198
拌:25292 伴:20276 瓣:29923 半:21322 办:21150 绊:32458 邦:37030
帮:24110 梆:26758 榜:27036 膀:33152 绑:32465 棒:26834 磅:30917
蚌:34444 镑:38225 傍:20621 谤:35876 苞:33502 胞:32990 包:21253
褒:35090 剥:21093 薄:34180 雹:38649 保:20445 堡:22561 饱:39281
宝:23453 抱:25265 报:25253 暴:26292 豹:35961 鲍:40077 爆:29190
杯:26479 碑:30865 悲:24754 卑:21329 北:21271 辈:36744 背:32972
贝:36125 钡:38049 倍:20493 狈:29384 备:22791 惫:24811 焙:28953
被:34987 奔:22868 苯:33519 本:26412 笨:31528 崩:23849 绷:32503
甭:29997 泵:27893 蹦:36454 迸:36856 逼:36924 鼻:40763 比:27604
鄙:37145 笔:31508 彼:24444 碧:30887 蓖:34006 蔽:34109 毕:27605
毙:27609 毖:27606 币:24065 庇:24199 痹:30201 闭:38381 敝:25949
弊:24330 必:24517 辟:36767 壁:22721 臂:33218 避:36991 陛:38491

原创技术方案—单字符摘要算法

第二版算法：

由于常见汉字Unicode码间隔没有规律，所以我们准备把汉字Unicode码的后几位留下，转成十进作为index去常见汉字中取词。这样就解决在知道算法的情况下，反解出现非常见字的问题，干扰项就不再容易排除了。

汉字Unicode留下后几位，涉及到一个模糊精确度和反解密复杂度的一个关系，模糊精确度高了相应的解密难度就降低了。

下面我们看一下常见汉字在我们算法下的碰撞程度：

1.汉字mask = 0b0011_1111_1111时

```
1:1、1:2、1:n统计: {1=100, 2=188, 3=183, 4=208, 5=157, 6=100, 7=47, 8=16, 9=3}
1对1映射汉字: [急, 逮, 怯, 框, 硝, 衡, 恢, 恤, 碴, 胆, 棉, 椅, 椎, 夜, 褥, 央, 橡,
煤, 熟, 冠, 妥, 冤, 冯, 委, 几, 姬, 称, 凹, 白, 娥, 判, 别, 稼, 剃, 打, 扦, 牢,
扦, 驻, 窃, 验, 助, 劲, 状, 骸, 押, 拨, 狰, 拳, 拴, 拽, 匈, 笋, 挚, 振, 欲, 茵,
筋, 议, 居, 摔, 汗, 汤, 莽, 瑶, 咆, 粘, 钠, 擒, 哩, 瓷, 锅, 蔓, 浓, 浩, 赵, 涕,
疮, 趴, 涸, 旁, 淀, 旗, 巾, 丹, 乙, 癖, 险, 些, 睁, 坊, 辑, 辰, 架, 螺, 达, 过,
染, 韦, 迪]
```

2.汉字mask = 0b0001_1111_1111时

```
1:1、1:2、1:n统计: {1=4, 2=8, 3=20, 4=36, 5=55, 6=73, 7=91,
8=70, 9=51, 10=49, 11=30, 12=15, 13=7, 14=3}
1对1映射汉字: [碴, 冠, 涸, 旗]
```

汉字尾数留10位和留9位。10位的查询精度会更高，因为10位的碰撞会小很多，但是1:1的字在知道算法和秘钥的的情况下是可以反推出原文的。9位的查询精度稍弱，因为9位的碰撞相对来说大一点，但是1:1的汉字较少。仔细观察会发现，不管是留10位还是留9位虽然我们改变了碰撞，但是由于汉字原本Unicode吗没有规律，导致分布非常不均衡，不能控制整体的一个碰撞概率。

原创技术方案—单字符摘要算法

第三版算法：

基于第二版发现分布不均衡的问题，我们把常用字乱序作为字典表。

1、加密文字先在乱序字典表中查找index，我们用取到的index下标代替没有规则Unicode码，如果非常用字还是使用Unicode码。（让参与计算的code尽量分布均衡）

2、第二步是和mask进行与运算丢失2位精度增加碰撞。

下面我们看一下常见汉字在我们算法下的碰撞程度：

1.汉字mask = 0b1111_1011_1101时

```
index总数: 936
1:1、1:2、1:n统计: {1=1, 2=7, 3=1, 4=927}
1对1映射汉字: [汁]
```

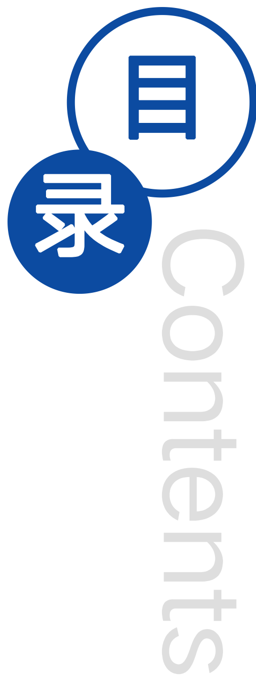
2.汉字mask = 0b0111_1011_1101时

```
index总数: 512
1:1、1:2、1:n统计: {4=88, 5=1, 6=7, 7=1, 8=415}
1对1映射汉字: []
```

Mask选择1的时候，可以看到碰撞分布相对来说集中在1:4上，此时我们只需要调整丢失精度的个数就能控制碰撞是1:2还是1:4还是1:8了。

Mask选择1的时候可能有一个常见字在知道算法和秘钥的情况下，因为此时我们计算的是常见字的碰撞程度，如果加上汉字16位二进制前面丢失的4位，情况就变成了2^5=32种情况。由于我们加密是整段文字，即使反推出个别字对于整体安全性影响不大，不会造成大批量数据泄密，同时前提是要知道算法、秘钥、delta和字典才具备反推的可能。

目录



1. 背景
2. 业内通行技术方案
3. 实现方案
4. 集成使用示例

实现方案

方案简介

- 目标

  1、为了方便业务线新项目的隐私加密要求，加快新项目的研发速度，降低研发成本。

  2、尽量降低业务线老项目的隐私加密改造成本，降低工作量，尽量少改动甚至不改动代码实现隐私加密。

- 实现内容

  1、新增数据时，字段自动加密后存储

  2、修改数据是，字段自动加密后修改

  3、实现加密后模糊查询列

  4、where条件中equal需要按照密文列加密算法加密内容后和密文列对比

  5、where条件中like需要按照单字符摘要算法加密内容后和模糊列进行like

实现方案—Querydsl

Querydsl如何实现字段加解密

- Querydsl简介

Querydsl仅仅是一个通用的查询框架，专注于通过Java API构建类型安全的SQL查询。

Querydsl可以通过一组通用的查询API为用户构建出适合不同类型ORM框架或者是SQL的查询语句，也就是说Querydsl是基于各种ORM框架以及SQL之上的一个通用的查询框架。

- Querydsl使用示例

```java
@PostMapping("/user")
@Transactional(rollbackFor = Exception.class)
public long insert(@RequestBody User user) {
    return sqlQueryFactoryAlter.insert(qUser).populate(user).execute();
}

@DeleteMapping("/user/{id}")
@Transactional(rollbackFor = Exception.class)
public long delete(@PathVariable Long id) {
    return sqlQueryFactoryAlter.delete(qUser).where(qUser.id.eq(id)
            .and(qUser.name.like( str: "熊%"))
            .and(qUser.phone.like( str: "130%"))
    ).execute();
}
```

```java
@PutMapping("/user/{id}")
@Transactional(rollbackFor = Exception.class)
public long put(@PathVariable Long id, @RequestBody User user) {
    return sqlQueryFactoryAlter.update(qUser).populate(user)
            .where(qUser.id.eq(id)
                    .and(qUser.name.like( str: "%熊%"))
            ).execute();
}

@GetMapping("/user/{id}")
public User getOne(@PathVariable Long id) {
    return sqlQueryFactoryAlter.select(qUser).from(qUser)
            .where(qUser.id.eq(id)
            .and(qUser.name.like( str: "熊%"))).fetchOne();
}
```

实现方案—Querydsl

Querydsl如何实现字段加解密

- 实现加解密原理

由于Querydsl是通过Java API构建类型安全的SQL查询，所以我们只要能在Bean对象中标记需要加密字段，就能有办法去实现自动加解密和模糊查询。

我们在medical_boot里面已经内置了Querydsl的这套隐私加密方案。

1. 定义加密Path集成StringPath

```
 * @date: 2022/3/21
 */
public class EncryptedStringPath extends StringPath {

    /***
     * 扩展字段Path
     */
    private StringPath extPath;    gxxiong2, 2022/3/21 16:20 • 新增querydsl扩

    /***
     * 是否启用扩展表
     */
    private boolean ext;

    public EncryptedStringPath(PathMetadata metadata, StringPath extPath) {
        super(metadata);
        this.extPath = extPath;
        if (extPath != null) {
            this.ext = true;
        }
    }

    public EncryptedStringPath(PathMetadata data) { super(data); }
```

实现方案—Querydsl

Querydsl如何实现字段加解密

2. Qbean标记需要加密字段

```java
/**
 * QTbPatientExt is a Querydsl query type for TbPatientExt
 */
public class QTbPatientExt extends com.querydsl.sql.RelationalPathBase<TbPatientExt> {

    private static final long serialVersionUID = -913188046;

    public static final QTbPatientExt qTbPatientExt = new QTbPatientExt( variable: "tb_patient_ext");

    public final StringPath createBy = createString( property: "createBy");

    public final StringPath patientName = this.add(
            new EncryptedStringPath(this.forProperty("patientName"), this.patientNameExt));

    public final StringPath patientNameExt = createString( property: "patientNameExt");

    public final DateTimePath<java.util.Date> createTime = createDateTime( property: "createTime", jav

    public final NumberPath<Integer> deleted = createNumber( property: "deleted", Integer.class);

    public final SimplePath<Object> extJson = createSimple( property: "extJson", Object.class);

    public final NumberPath<Long> id = createNumber( property: "id", Long.class);
```

3. 在插入和更新方法处统一加密

```java
public void serializeInsert(QueryMetadata metadata,
    for (SQLInsertBatch batch : batches) {
        List<Path<?>> columns = batch.getColumns();
        List<Expression<?>> values = batch.getValue
        hackInsertExtra(columns, values);

public void serializeUpdate(QueryMetadata metadata, Relat
    hackUpdateExtra(updates);
    super.serializeUpdate(metadata, entity, updates);

lic void hackInsertExtra(List<Path<?>> columns, List<Expression<?>> values) {
List<Path<?>> extraColumns = new ArrayList<>();
List<Expression<?>> extraValues = new ArrayList<>();
for (int i = 0; i < columns.size(); i++) {
    Path path = columns.get(i);
    Expression value = values.get(i);
    if (path instanceof EncryptedStringPath && value instanceof Constant) {
        Path extraPath = ((EncryptedStringPath) path).getExtPath();
        Object constant = ((Constant) value).getConstant();
        if (constant instanceof Null) {
            values.set(i, ConstantImpl.create(constant));
        } else {
            values.set(i, ConstantImpl.create(ENCRYPTION.encrypt(constant.to
        }
        if (extraPath != null) {
            extraColumns.add(extraPath);
            if (constant instanceof Null) {
                extraValues.add(ConstantImpl.create(constant));
            } else {
                extraValues.add(ConstantImpl.create(ENCRYPTIONEXT.mask(const
            }
    }
```

实现方案—Querydsl

Querydsl如何实现字段加解密

4. 查询的时候字段解密

```java
public class EncryptedStringType extends AbstractType<String> {

    public EncryptedStringType() { super(12); }

    public EncryptedStringType(int type) { super(type); }

    @Override
    public String getValue(ResultSet rs, int startIndex) throws
        return ENCRYPTION.decrypt(rs.getString(startIndex));
    }

    @Override
    public void setValue(PreparedStatement st, int startIndex,
        st.setString(startIndex, value);
    }

    @Override
    public Class<String> getReturnedClass() { return String.cla

}
```

5. 加密字段like、contains、eq等情形分别处理

```java
@Override
public BooleanExpression eq(String right) {
    if (ext) {
        return extPath.eq(right);
    } else {
        return super.eq(ENCRYPTION.encrypt(right));
    }
}

@Override
public BooleanExpression like(String str) {
    if (ext) {
        return extPath.like(str);
    } else {
        Map<String, String> preLikeMap = prepareLike(str);
        StringBuffer sb = new StringBuffer();
        str = sb.append(preLikeMap.get("start")).append(EN
        return super.like(str);
    }
}

@Override
public BooleanExpression contains(String str) {
    if (ext) {
        return extPath.contains(str);
    } else {
        return super.contains(ENCRYPTION.encrypt(str));
    }
}

@Override
public BooleanExpression startsWith(String str) {
```

实现方案—Mybatis

Mybatis框架下如何解决

- 自定义拦截器和注解

Mybatis的Interceptor（拦截器），可以用来拦截SQL的参数处理、SQL语句构建、SQL执行和返回结果处理等阶段。要实现字段的加解密，可以通过**自定义两个拦截器**：参数处理拦截器和结果集拦截器，在参数处理拦截器中对字段进行加密，在结果集拦截器中对字段进行解密。

限制：

该方案类似实现起来要比较麻烦，而且只能解决注解标记的对象操作。

思路：

是否能将SQL解析变成代码可以处理的东西，然后根据规则进行处理重新组装成SQL执行，这样就不再局限于框架和对象。我们发现AST（抽象语法树）刚好就是来处理这个事情的，大家经常用到的druid连接池就用到了AST抽象语法树（用来实现防御SQL注入、合并统计没有参数化的、 SQL格式化、分库分表）。

实现方案一 AST介绍

AST（抽象语法树）

解析过程分为词法解析和语法解析。 词法解析器用于将 SQL 拆解为不可再分的原子符号，称为 Token。并根据不同数据库方言所提供的字典，将其归类为关键字，表达式，字面量和操作符。 再使用语法解析器将词法解析器的输出转换为抽象语法树。

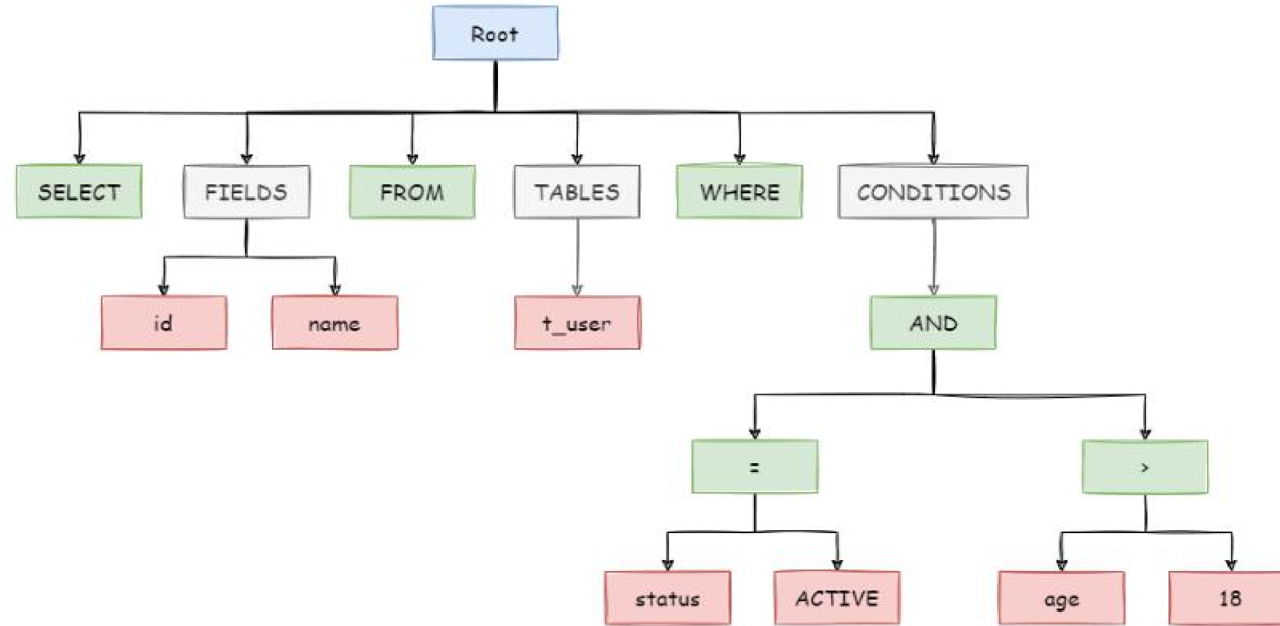
例如，以下 SQL：

```
SELECT id, name FROM t_user WHERE status = 'ACTIVE' AND age > 18
```

解析之后的为抽象语法树见下图。

实现方案一 AST介绍

AST（抽象语法树）



抽象语法树中的关键字的 Token 用绿色表示，变量的 Token 用红色表示，灰色表示需要进一步拆分。

最后，通过 visitor 对抽象语法树遍历构造域模型，通过域模型（SQLStatement）去提炼分片所需的上下文，并标记有可能需要改写的位置。供分片使用的解析上下文包含查询选择项（Select Items）、表信息（Table）、分片条件（Sharding Condition）、自增主键信息（Auto increment Primary Key）、排序信息（Order By）、分组信息（Group By）以及分页信息（Limit、Rownum、Top）。SQL 的一次解析过程是不可逆的，一个个 Token 按 SQL 原本的顺序依次进行解析，性能很高。考虑到各种数据库 SQL 方言的异同，在解析模块提供了各类数据库的 SQL 方言字典。

实现方案— AST介绍

AST（抽象语法树）

SQLStatement生成

```
public SQLStatement parse() {
      //解析获取抽象语法树AST
       SQLAST ast = parserEngine.parse();
      //从语法树中获取sql 片段
       Collection<SQLSegment> sqlSegments = extractorEngine.extract(ast);
       Map<ParserRuleContext, Integer> parameterMarkerIndexes = ast.getParameterMarkerIndexes();
      //将解析获取的语法树遍历填充在SQLStatement对象中返回，供后续路由使用
       return fillerEngine.fill(sqlSegments, parameterMarkerIndexes.size(), ast.getSqlStatementRule());
   }
```

上边提到的sql片段即SQLSegment，其实是定义的一种数据结构，是对词法解析中每一个Token的封装。

```
public class ColumnSegment implements SQLSegment, PredicateRightValue, OwnerAvailable<TableSegment> {
    //在sql中开始index
    private final int startIndex;
    //结束index
    private final int stopIndex;
    //字段名column
    private final String name;
    //是否有 `，'等，比如mysql中查询中 select `id`,`name` from xxx
    private final QuoteCharacter quoteCharacter;
    //字段属于哪个表
    private TableSegment owner;
}
```

实现方案— AST介绍

AST（抽象语法树）

SQLSegment封装到SQLStatement中，生成抽象语法树

```
public SQLStatement fill(final Collection<SQLSegment> sqlSegments, final int parameterMarkerCount, final SQLStatementRule rule) {
        //根据语法规则中配置的class name获取实现类，比如有SQLSelectStatement 、SQLinsertStatement等
        SQLStatement result = rule.getSqlStatementClass().newInstance();
        Preconditions.checkArgument(result instanceof AbstractSQLStatement, "%s must extends AbstractSQLStatement", result.getClass().getName());
        ((AbstractSQLStatement) result).setParametersCount(parameterMarkerCount);
        result.getAllSQLSegments().addAll(sqlSegments);
        //对应的SQLSegment实现了filter，fiter作用是将有关键的SQLSegment放在对应的数据结构中，比如有orderby 或者
        for (SQLSegment each : sqlSegments) {
            Optional<SQLSegmentFiller> filler = parseRuleRegistry.findSQLSegmentFiller(databaseType, each.getClass());
            if (filler.isPresent()) {
                filler.get().fill(each, result);
            }
        }
        return result;
    }
```

图右是OrderByFiller的实现

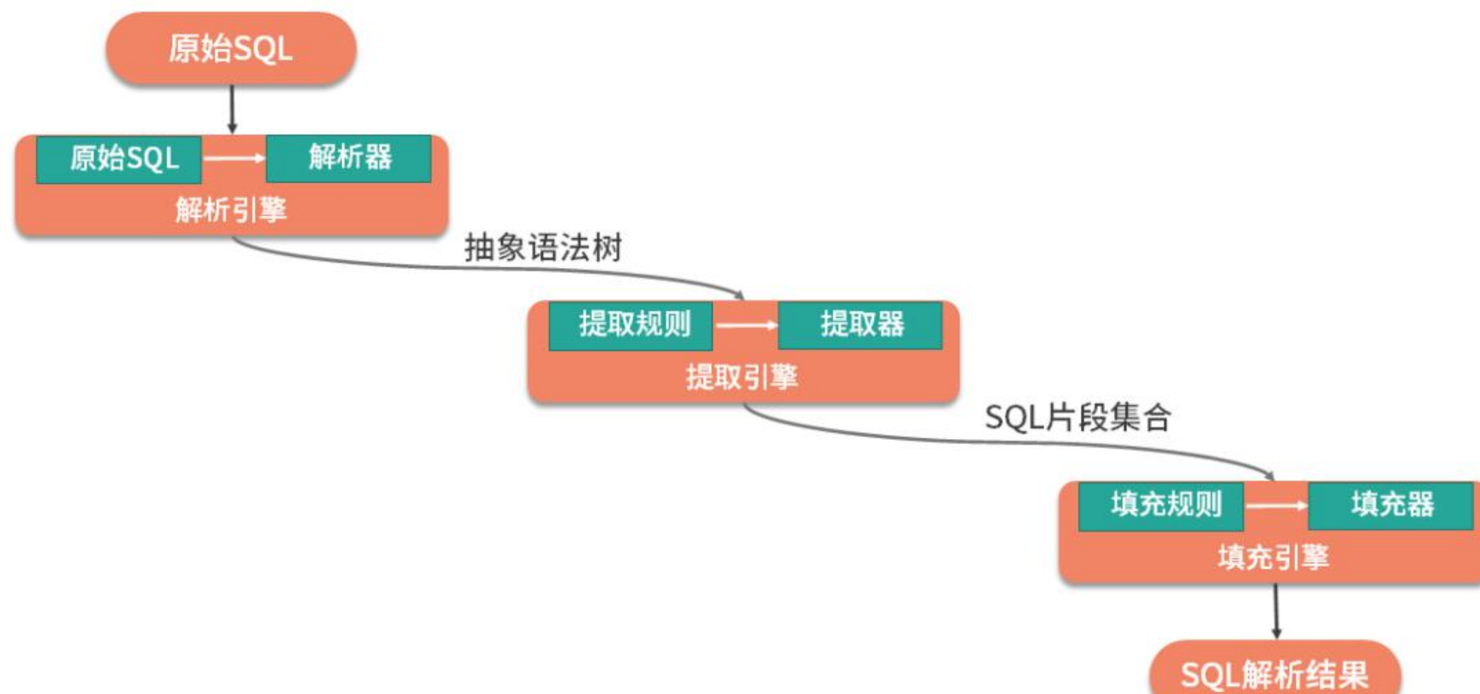
```
public final class OrderByFiller implements SQLSegmentFiller<OrderBySegment> {

    @Override
    public void fill(final OrderBySegment sqlSegment, final SQLStatement sqlStatement) {
        ((SelectStatement) sqlStatement).setOrderBy(sqlSegment);
    }
}
```

实现方案— AST介绍

AST（抽象语法树）

生成 SQL-AST

主要包括三个步骤：
（1）通过 SQLParserEngine 生成 SQL 抽象语法树。
（2）通过 SQLSegmentsExtractorEngine 提取 SQLSegment。
（3）通过 SQLStatementFiller 填充 SQLStatement。

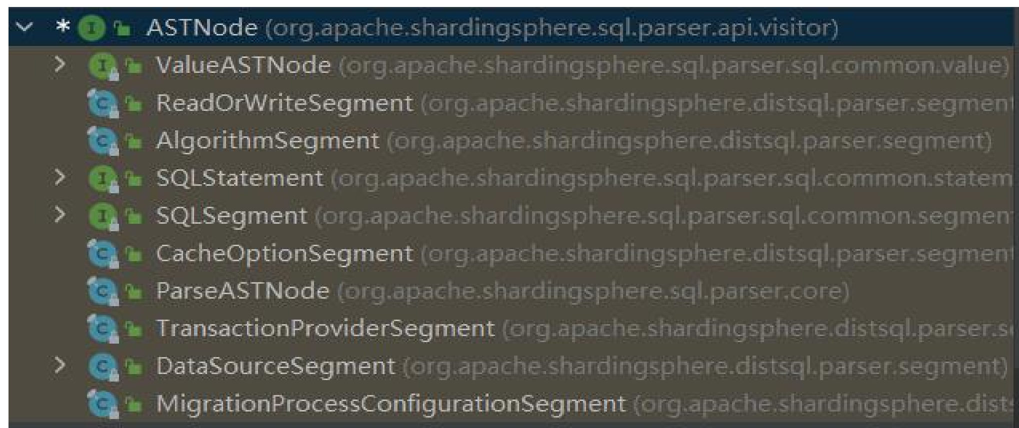


实现方案— AST介绍

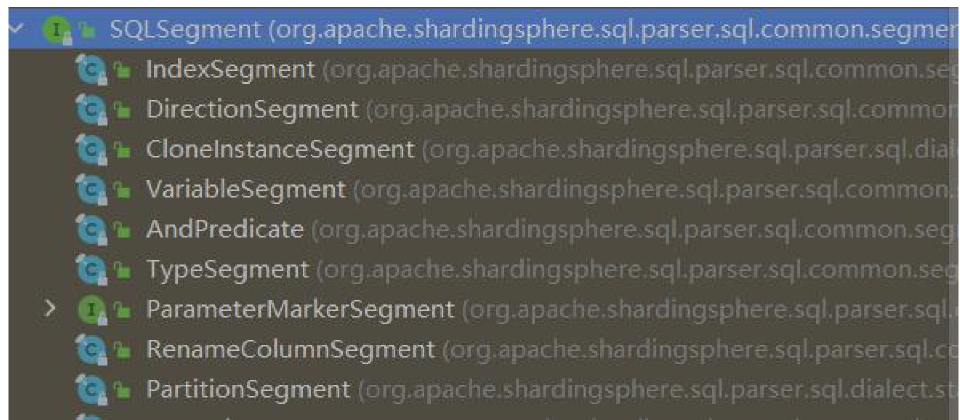
AST（抽象语法树）

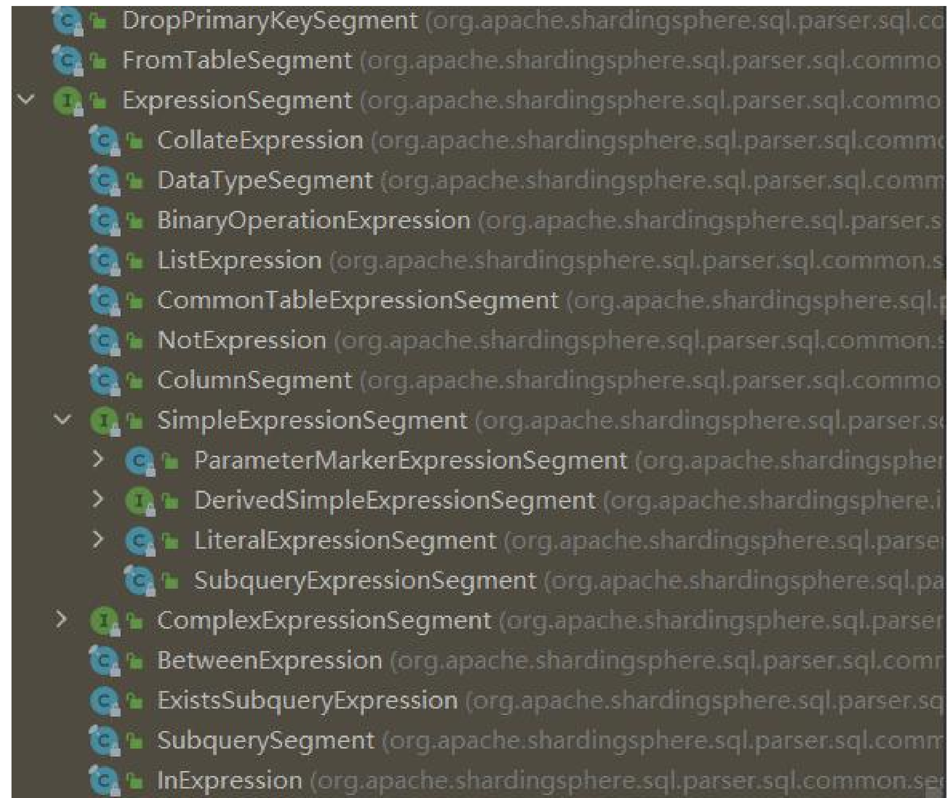
ASTNode

语法树结构：



SQLSegment结构：





这里截取了部分语法树的结构，可以看出语法树的结构是十分复杂的。

实现方案— AST调研

AST调研

前面介绍了AST，然后我们准备朝着SQl解析改写的思路去做加解密和查询改写，调研到Druid的SQL Parser。

- Druid SQL Parser
  1. 通过代码的方式实现词法解析器和语法解析器
  2. 支持多种DB: Mysql, PostgreSQL, SQLServer, Oracle, odps, db2, hive, SQL92
  3. SQL 格式化，SQL参数化
  4. 支持自定义visitor

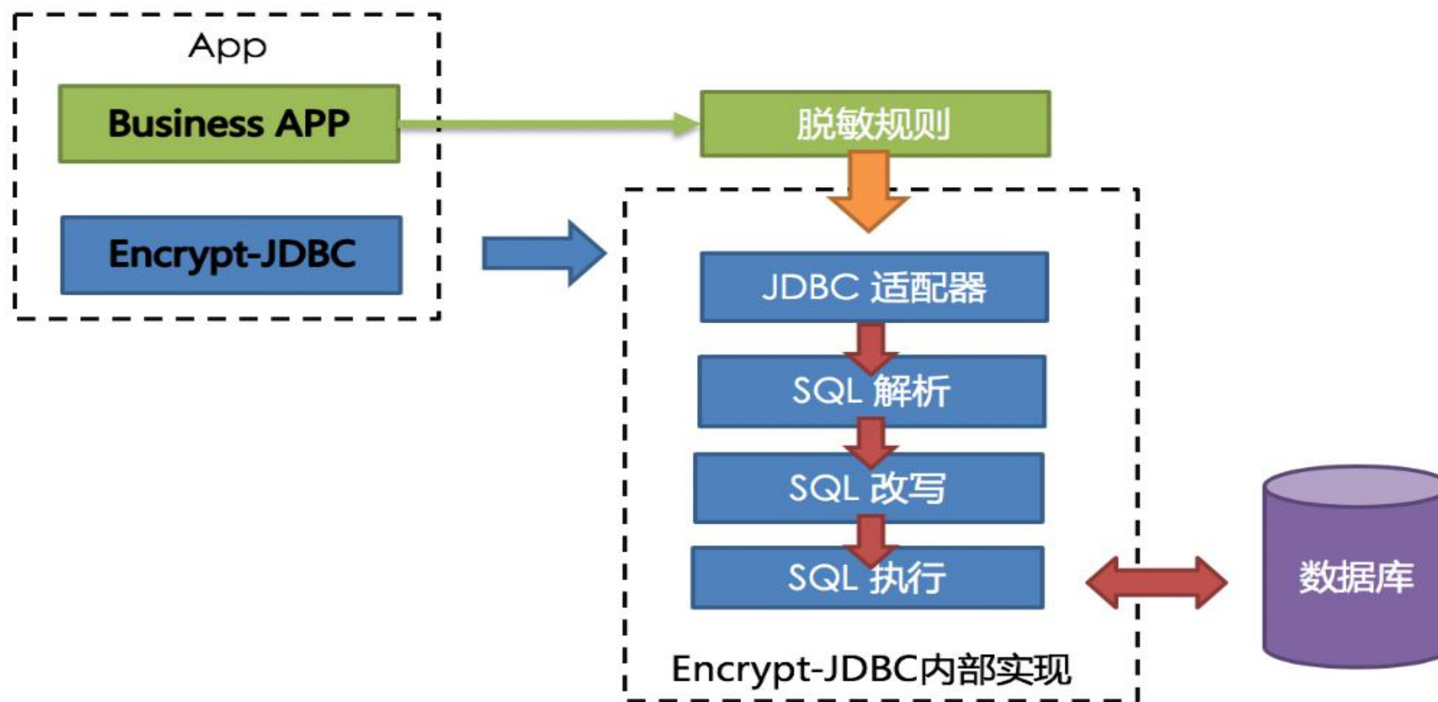
该方案可以需要自己定义visitor去实现各种场景下SQL的加解密和字段改写，工作量和测试量非常大。此时我们发现ShardingSphere-JDBC正是按照这个思路内置实现了数据加密模块。

*ShardingSphere-JDBC早期为了追求性能和快速实现功能的时候，采用的正是 Druid 作为 SQL 解析器。经实际测试，它的性能远超其它解析器，后面为了方便扩展采取了可配置的ANTLR作为解析引擎。

实现方案一 AST调研

ShardingSphere-JDBC数据加密调研

ShardingSphere 通过对用户输入的 SQL 进行解析，并依据用户提供的加密规则对 SQL 进行改写，从而实现对原文数据进行加密，并将原文数据（可选）及密文数据同时存储到底层数据库。 在用户查询数据时，它仅从数据库中取出密文数据，并对其解密，最终将解密后的原始数据返回给用户。 ShardingSphere 自动化 & 透明化了数据加密过程，让用户无需关注数据加密的实现细节，像使用普通数据那样使用加密数据。

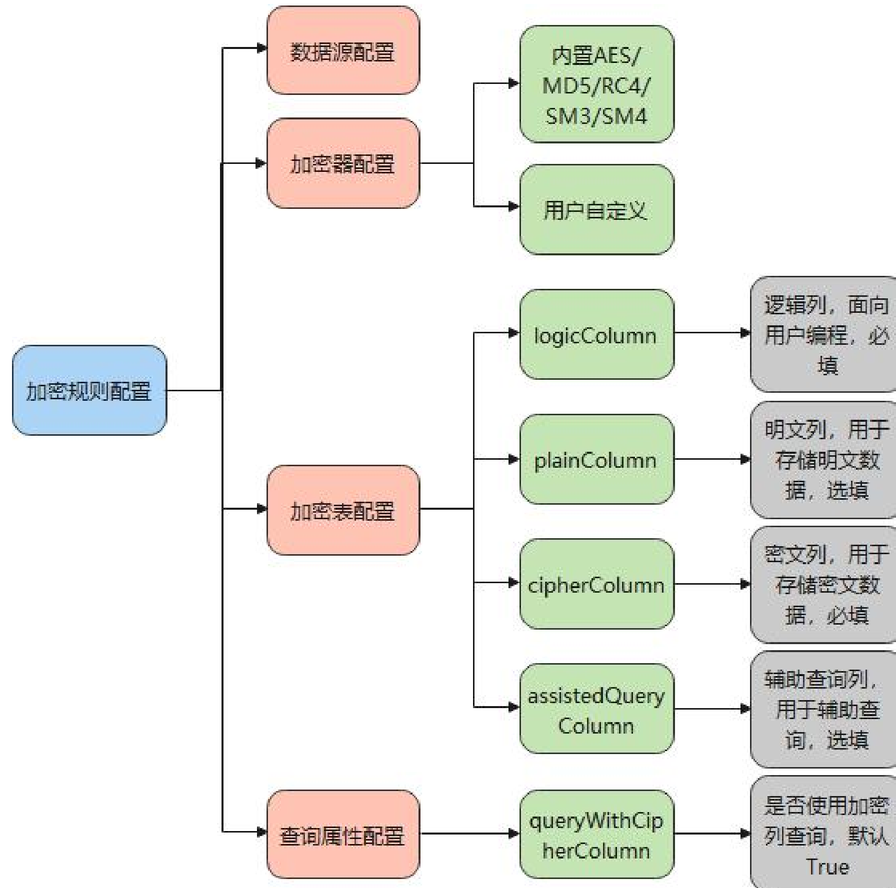


实现方案一 AST调研

ShardingSphere-JDBC数据加密调研

- 加密规则

加密配置主要分为四部分：数据源配置，加密算法配置，加密表配置以及查询属性配置，其详情如下图所示：



- **加密算法配置**：指使用什么加密算法进行加解密。目前 ShardingSphere 内置了五种加解密算法：AES，MD5，RC4，SM3 和 SM4。用户还可以通过实现 ShardingSphere 提供的接口，自行实现一套加解密算法。
- **logicColumn列的意义**：最终目的是希望屏蔽底层对数据的加密处理，不需要让用户知道数据是如何被加解密的、如何将明文数据存储到 plainColumn，将密文数据存储到 cipherColumn，将辅助查询数据存储到 assistedQueryColumn。

实现方案— AST调研

ShardingSphere-JDBC数据加密调研

- 加密、转换流程

1、插入时实现：插入列自动增加字段，原文按照相应加密规则加密插入

2、更新时实现：修改列自动增加字段，原文按照相应加密规则加密修改，where语句自动改写

3、查询时实现：where语句自动改写，结果自动解密

缺点：where加密字段常见操作实现了eq，不支持like查询。

实现方案一 ShardingSphere-JDBC二次开发

ShardingSphere-JDBC二次开发

- 思路

How to support the fuzzy query of encrypted fields #20435

Open cheese8 opened this issue on 23 Aug · 7 comments

cheese8 commented on 23 Aug • edited Contributor

Solutions: @strongduanmu

1. Homomorphism algorithm, but there is no good homomorphic encryption algorithm implemented for now.
2. Besides the cipherColumn and assistedQueryColumn columns, add additional fuzzyColumn and fuzzyAlgorithm which works like the cipherColumn and assistedQueryColumn columns, rewrite `LIKE plainColumn` into `LIKE fuzzyColumn` to meet user requirements. The `API` changed as below:

```
public final class EncryptColumnRuleConfiguration {

    private final String logicColumn;

    private final String cipherColumn;

    private final String assistedQueryColumn;

    private final String plainColumn;

    private final String encryptorName;

    private final String assistedQueryEncryptorName;

    private final Boolean queryWithCipherColumn;

    // new added
    private final String fuzzyColumn;

    // new added
    private final String fuzzyEncryptorName;
```

https://github.com/apache/shardingsphere/issues/20435

liming0 commented on 13 Jul • edited by terrymanu Author

> I have no idea for how to support `LIKE`, any suggestions?

I think we only need to allow the like keyword to filter the encrypted column. The specific implementation of encryption and query can be realized by users themselves

liming0 commented on 13 Jul • edited by terrymanu Author

> I have no idea for how to support `LIKE`, any suggestions?

My idea now is to divide the content into groups of X characters, then encrypt each group separately, and then splice them together for storage. When querying, repeat the steps of grouping encryption, and then query; At present, there are two points to be controlled in this method: first, the encrypted characters will be relatively long. This method uses storage space to realize this requirement; Second, we need to control the size of variable x, which should not be too short. Too long will also affect the search experience

terrymanu added type: discussion feature: encrypt labels on 25 Jul

terrymanu commented on 25 Jul • edited Member

How about the performance and the storage capacity? Any idea to deal with like `%`?

https://github.com/apache/shardingsphere/issues/19026

1、第一个Issues是ShardingSphere官方的一个提议增加一个模糊查询列，但由于没有好的like查询算法在收集意见。

2、第二个Issues有人提议用分词组合加密存储去实现like查询。

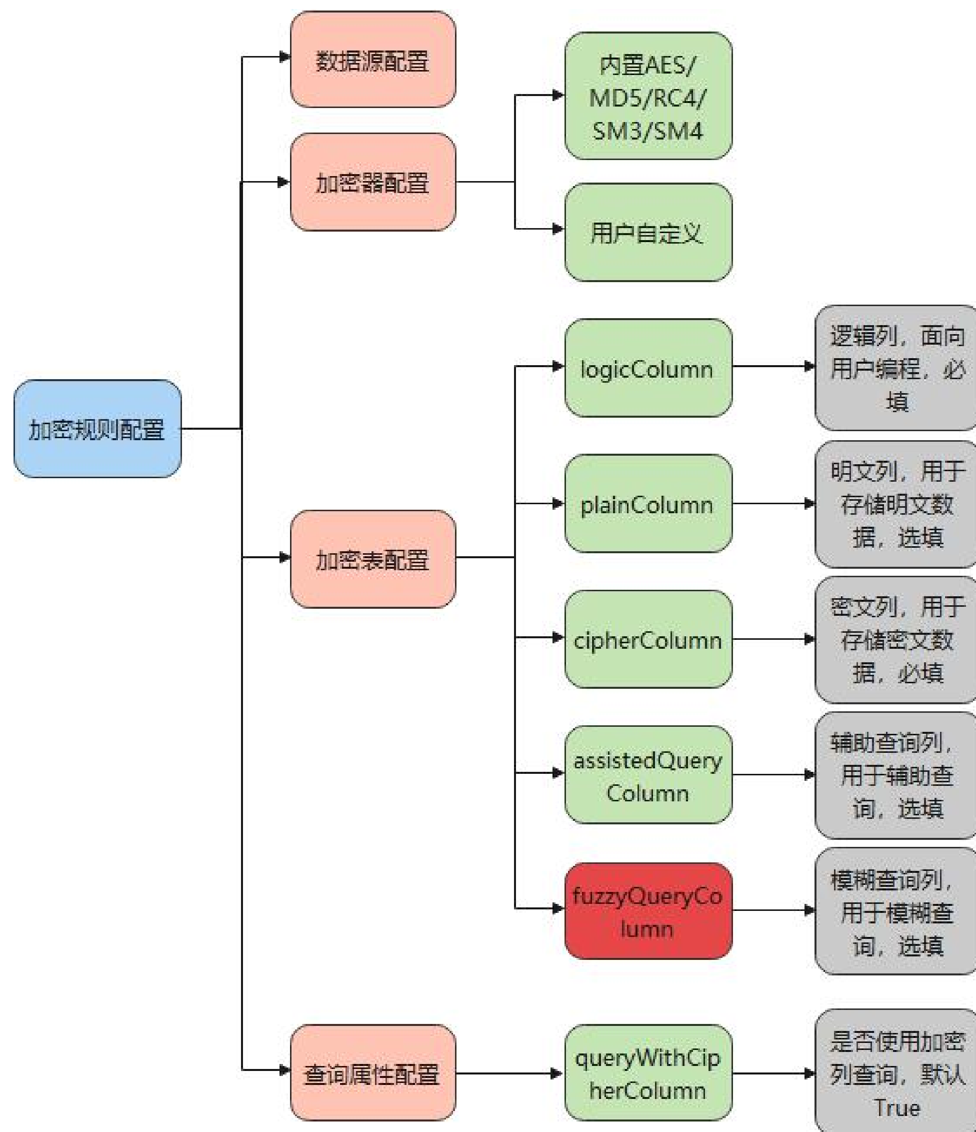
实现方案一 ShardingSphere-JDBC二次开发

ShardingSphere-JDBC二次开发

- 思路

　　分词组合加密存储的算法去实现模糊查询，和原文相比会带来巨大的数据存储压力和性能损耗。

　　前面我们介绍了单字符摘要算法，可以解决like查询的问题，还不会带来分词组合庞大的数据压力，所以我们准备按照官方的这个思路增加fuzzyQueryColumn列用来实现模糊查询。



实现方案一 ShardingSphere-JDBC二次开发

ShardingSphere-JDBC二次开发

- SQL改写

辅助列字段增加后，我们就需要关注如何改写SQL。

1 、先看核心类KernelProcessor

```
public final class KernelProcessor {
    public KernelProcessor() {
    }

    public ExecutionContext generateExecutionContext(QueryContext queryContext, Shar
        RouteContext routeContext = this.route(queryContext, database, props, connec
        SQLRewriteResult rewriteResult = this.rewrite(queryContext, database, globa
        ExecutionContext result = this.createExecutionContext(queryContext, databas
        this.logSQL(queryContext, props, result);
        return result;
    }

    private RouteContext route(QueryContext queryContext, ShardingSphereDatabase dat
        return (new SQLRouteEngine(database.getRuleMetaData().getRules(), props)).r
    }

    private SQLRewriteResult rewrite(QueryContext queryContext, ShardingSphereDataba
        SQLRewriteEntry sqlRewriteEntry = new SQLRewriteEntry(database, globalRuleMe
        return sqlRewriteEntry.rewrite(queryContext.getSql(), queryContext.getParame
    }
```

发现rewriter是个**SQLRewriteEntry**对象。

2、SQLRewriteEntry

```
public SQLRewriteResult rewrite(String sql, List<Object> parameters, 
    SQLRewriteContext sqlRewriteContext = this.createSQLRewriteContex
    SQLTranslatorRule rule = (SQLTranslatorRule)this.globalRuleMetaDa
    DatabaseType protocolType = this.database.getProtocolType();
    DatabaseType storageType = this.database.getResource().getDatabas
    return (SQLRewriteResult)(routeContext.getRouteUnits().isEmpty() 
}

private SQLRewriteContext createSQLRewriteContext(String sql, List<Obj
    SQLRewriteContext result = new SQLRewriteContext(this.database.ge
    this.decorate(this.decorators, result, routeContext);
    result.generateSQLTokens();
    return result;
}

private void decorate(Map<ShardingSphereRule, SQLRewriteContextDecora
    Iterator var4 = decorators.entrySet().iterator();

    while(var4.hasNext()) {
        Entry<ShardingSphereRule, SQLRewriteContextDecorator> entry =
        ((SQLRewriteContextDecorator)entry.getValue()).decorate((Shar
    }
```

1、初始SQL重写上下文SQLRewriteContext对象

2、 依次调用注册的SQLRewriteContextDecorator实现类的decorate方法，根据配置去改写SQL。

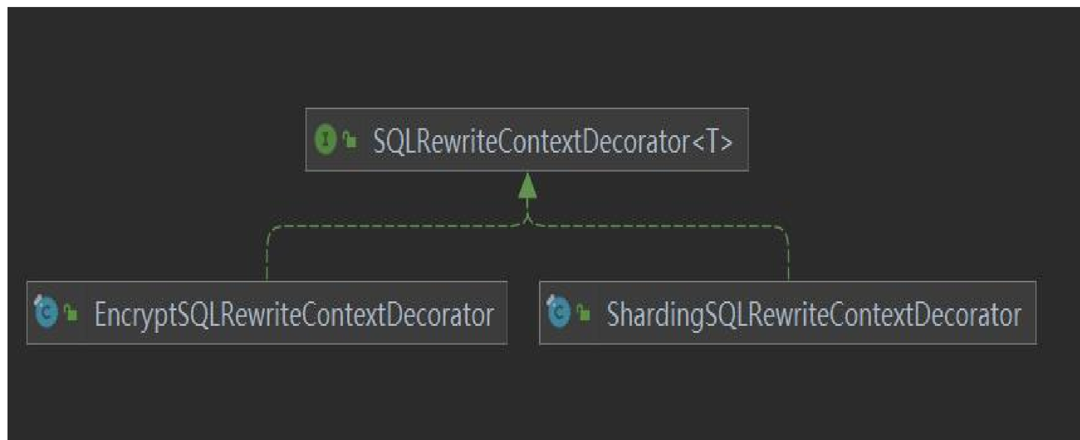
实现方案一 ShardingSphere-JDBC二次开发

ShardingSphere-JDBC二次开发

- SQL改写

  1、目前项目中已实现的SQLRewriteContextDecorate

  如图下所示：



分别对应数据分片SQL重写装饰器、加密SQL重写装饰器。

2、加密SQL重写装饰器

```java
@Override
public void decorate(final EncryptRule encryptRule, final ConfigurationProperties props, final SQLRewriteContext
    SQLStatementContext<?> sqlStatementContext = sqlRewriteContext.getSqlStatementContext();
    if (((CommonSQLStatementContext) sqlStatementContext).getSqlHintExtractor().isHintSkipEncryptRewrite() || !cc
        return;
    }
    Collection<EncryptCondition> encryptConditions = createEncryptConditions(encryptRule, sqlRewriteContext);
    encryptRule.setSchemaMetaData(sqlRewriteContext.getDatabaseName(), sqlRewriteContext.getSchemas());
    if (!sqlRewriteContext.getParameters().isEmpty()) {
        Collection<ParameterRewriter> parameterRewriters = new EncryptParameterRewriterBuilder(encryptRule,
                sqlRewriteContext.getDatabaseName(), sqlRewriteContext.getSchemas(), sqlStatementContext, encrypt
        rewriteParameters(sqlRewriteContext, parameterRewriters);
    }
    Collection<SQLTokenGenerator> sqlTokenGenerators = new EncryptTokenGenerateBuilder(encryptRule,
            sqlStatementContext, encryptConditions, sqlRewriteContext.getDatabaseName()).getSQLTokenGenerators();
    sqlRewriteContext.addSQLTokenGenerators(sqlTokenGenerators);
}   Charles, 2022/9/14 9:27 • init 5.2.0
```

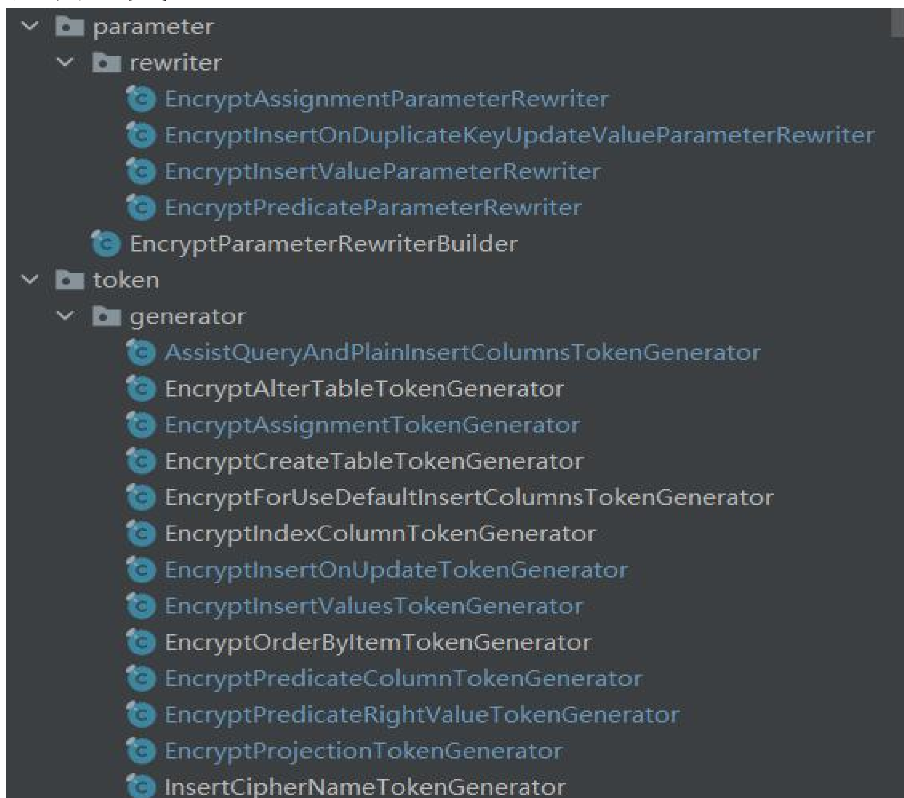
在decorate方法中，初始化EncryptParameterRewriterBuilder和EncryptTokenGenerateBuilder分两块去进行SQL改写

实现方案一 ShardingSphere-JDBC二次开发

ShardingSphere-JDBC二次开发

- SQL改写

1、改写类



根据以上分析，我们知道了需要改写的地方。下面介绍下简单Insert和查询where的改写

2、简单Insert改写

```
private void encryptInsertValue(final EncryptAlgorithm encryptAlgorithm, final EncryptAlgorithm assistEncryptor, final EncryptAlgorithm fuzzyE
                                final int parameterIndex, final Object originalValue, final StandardParameterBuilder parameterBuilder,
                                final EncryptContext encryptContext) {
    parameterBuilder.addReplacedParameters(parameterIndex, encryptAlgorithm.encrypt(originalValue, encryptContext));
    Collection<Object> addedParameters = new LinkedList<>();
    if (null != assistEncryptor) {...}
    //判断是否配置模糊列，如果有就按照配置的算法对值进行加密放入sql    You, Yesterday • Uncommitted changes
    if (null != fuzzyEncryptor) {
        Optional<String> fuzzyColumnName = encryptRule.findFuzzyColumn(encryptContext.getTableName(), encryptContext.getColumnName());
        Preconditions.checkArgument(fuzzyColumnName.isPresent(), errorMessage: "Can not find fuzzy query Column Name");
        addedParameters.add(fuzzyEncryptor.encrypt(originalValue, encryptContext));
    }
    if (encryptRule.findPlainColumn(encryptContext.getTableName(), encryptContext.getColumnName()).isPresent()) {
        addedParameters.add(originalValue);
    }
    if (!addedParameters.isEmpty()) {
        if (!parameterBuilder.getAddedIndexAndParameters().containsKey(parameterIndex)) {
            parameterBuilder.getAddedIndexAndParameters().put(parameterIndex, new LinkedList<>());
        }
        parameterBuilder.getAddedIndexAndParameters().get(parameterIndex).addAll(addedParameters);
```

EncryptInsertValueParameterRewriter类中的encryptInsertValue方法判断是否配置模糊列，调用配置的算法给输入值加密。

实现方案一 ShardingSphere-JDBC二次开发

ShardingSphere-JDBC二次开发

- SQL改写

1、简单Insert改写

```java
private void encryptToken(final InsertValue insertValueToken, final String schemaName, final String tableName,
                          final InsertStatementContext insertStatementContext, final InsertValueContext insertValueContext
    Optional<SQLToken> useDefaultInsertColumnsToken = findPreviousSQLToken(UseDefaultInsertColumnsToken.class);
    Iterator<String> descendingColumnNames = insertStatementContext.getDescendingColumnNames();
    while (descendingColumnNames.hasNext()) {
        String columnName = descendingColumnNames.next();
        Optional<EncryptAlgorithm> encryptor = encryptRule.findEncryptor(tableName, columnName);
        if (encryptor.isPresent()) {
            int columnIndex = useDefaultInsertColumnsToken.map(optional ->
                    ((UseDefaultInsertColumnsToken) optional).getColumns().indexOf(columnName))
                .orElseGet(() -> insertStatementContext.getColumnNames().indexOf(columnName));
            Object originalValue = insertValueContext.getLiteralValue(columnIndex).orElseThrow(
                    () -> new UnsupportedEncryptInsertValueException(columnIndex));
            EncryptContext encryptContext = EncryptContextBuilder.build(databaseName, schemaName, tableName, columnName);
            addPlainColumn(insertValueToken, columnIndex, encryptContext, insertValueContext, originalValue);
            int index = 1;
            if (encryptRule.findAssistedQueryEncryptor(tableName, columnName).isPresent()) {
                addAssistedQueryColumn(insertValueToken, encryptRule.findAssistedQueryEncryptor(tableName,
                        columnName).get(), columnIndex, encryptContext, insertValueContext, originalValue);
                index = index + 1;
            }
            if (encryptRule.findFuzzyEncryptor(tableName, columnName).isPresent()) {
                addFuzzyQueryColumn(insertValueToken, encryptRule.findFuzzyEncryptor(tableName, columnName).get(),
                        columnIndex, encryptContext, insertValueContext, originalValue, index);
            }
```

EncryptInsertValuesTokenGenerator类中的encryptToken方法改写SQL增加Fuzzy字段名。

2、简单select where改写

```java
private List<Object> getEncryptedValues(
        final String schemaName, final EncryptCondition encryptCondition,
        final List<Object> originalValues) {
    String tableName = encryptCondition.getTableName();
    String columnName = encryptCondition.getColumnName();
    // like 优先处理
    if (encryptCondition instanceof EncryptLikeCondition &&
            encryptRule.findFuzzyColumn(tableName, columnName).isPresent()) {
        return encryptRule.getEncryptFuzzyValues(databaseName,
                schemaName, tableName, columnName, originalValues);
    }

    return encryptRule.findAssistedQueryColumn(tableName, columnName).isPresent
            ? encryptRule.getEncryptAssistedQueryValues(databaseName, schemaNam
            : encryptRule.getEncryptValues(databaseName, schemaName, tableName,
```

EncryptPredicateParameterRewriter类中的getEncryptedValues方法判断是否配置模糊列并且是like查询，调用配置的算法给查询值加密。

实现方案一 ShardingSphere-JDBC二次开发

ShardingSphere-JDBC二次开发

- SQL改写

  1、简单select where改写

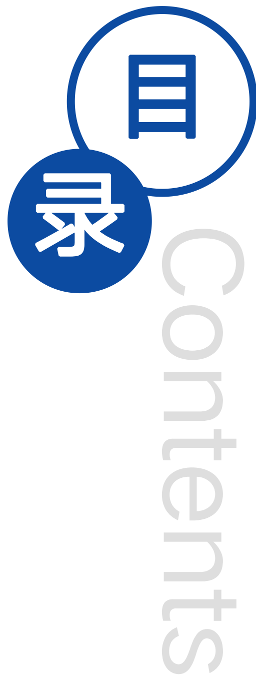
```java
private Collection<SubstitutableColumnNameToken> generateSQLTokens(
        final Collection<ColumnSegment> columnSegments,
        final Map<String, String> columnExpressionTableNames,
        final Collection<WhereSegment> whereSegments) {
    Collection<SubstitutableColumnNameToken> result = new LinkedHashSet<>();
    for (ColumnSegment each : columnSegments) {
        String tableName = Optional.ofNullable(columnExpressionTableNames.get(each.getExpressi
        Optional<EncryptTable> encryptTable = encryptRule.findEncryptTable(tableName);
        if (!encryptTable.isPresent() || !encryptTable.get().findEncryptColumn(each.getIdentif
        int startIndex = each.getOwner().isPresent() ? each.getOwner().get().getStopIndex() +
        int stopIndex = each.getStopIndex();
        boolean queryWithCipherColumn = encryptRule.isQueryWithCipherColumn(tableName, each.ge
        if (!queryWithCipherColumn) {...}
        // like 优先处理
        BinaryOperationExpression boe = findBinaryOperationExpression(whereSegments, each);
        if (boe != null && boe.getOperator().equalsIgnoreCase( anotherString: "LIKE")) {
            Optional<String> fuzzyColumn = encryptTable.get()
                    .findFuzzyColumn(each.getIdentifier().getValue());    You, A minute ago
            if (!fuzzyColumn.isPresent()) {
                throw new UnsupportedEncryptSQLException("LIKE");
            } else {
                result.add(new SubstitutableColumnNameToken(startIndex,
                        stopIndex, createColumnProjections(fuzzyColumn.get())));
                continue;
            }
        }
```

```java
private BinaryOperationExpression findBinaryOperationExpression(
        final Collection<WhereSegment> whereSegments,
        final ColumnSegment columnSegment) {
    for (WhereSegment whereSegment : whereSegments) {
        Collection<AndPredicate> andPredicates =
                ExpressionExtractUtil.getAndPredicates(whereSegment.getExpr());
        for (AndPredicate andPredicate : andPredicates) {
            for (ExpressionSegment predicate : andPredicate.getPredicates()) {
                if (predicate instanceof BinaryOperationExpression) {
                    BinaryOperationExpression boe = (BinaryOperationExpression) predicate;
                    if (columnMatch(columnSegment, boe.getLeft())) {
                        return boe;
                    }
                }
            }
        }
    }
    return null;
```

EncryptPredicateColumnTokenGenerator类中的generateSQLTokens方法判断是否配置模糊列并且是like查询，更改SQL Like为配置的模糊列。

这里就用到了抽象语法树中的BinaryOperationExpression去获取是否为Like类型。

目录



1. 背景

2. 业内通行技术方案

3. 实现方案

4. 集成使用示例

集成使用示例— ShardingSphere-JDBC使用示例

ShardingSphere-JDBC使用示例

- pom文件配置

```xml
<dependency>
    <groupId>org.apache.shardingsphere</groupId>
    <artifactId>shardingsphere-jdbc-core-spring-boot-starter</artifactId>
</dependency>
```

Pom文件引入包即可，版本号后期会加入到medical-boot管理

- yml配置

```yaml
encrypt:
  encryptors:
    fuzzy-encryptor:
      type: FUZZY
    name-encryptor:
      type: AES
      props:
        aes-key-value: 123456abc
```

配置好需要用到的算法，有官方内置的AES、MD5、RC4、SM3、SM4和我们的FUZZY算法

```yaml
tables:
  user:
    columns:
      name:  # 加密列名称
        cipherColumn: name # 加密列名称
        encryptorName: name-encryptor # 加密算法名称（名称不能有下划线）
        assistedQueryColumn: name_ext
        assistedQueryEncryptorName: name-encryptor
        fuzzyQueryColumn: name_fuzzy
        fuzzyQueryEncryptorName: fuzzy-encryptor
      phone: # 加密列名称
        cipherColumn: phone # 加密列名称
        encryptorName: name-encryptor # 加密算法名称（名称不能有下划线）
        fuzzyQueryColumn: phone_fuzzy
        fuzzyQueryEncryptorName: fuzzy-encryptor
queryWithCipherColumn: true # 是否使用加密列进行查询. 在有原文列的情况下, 可以使用原文列进行查询
```

1、配置好自己需要加密的表、列

2、选择加密列和模糊列以及刚才配置好的加密方案名

3、配置是否使用加密列查询，默认是true

集成使用示例— ShardingSphere-JDBC使用示例

ShardingSphere-JDBC使用示例

- Insert示例

```
@PostMapping("/user")
@Transactional(rollbackFor = Exception.class)
public long insert(@RequestBody User user) {
    return sqlQueryFactoryAlter.insert(qUser).populate(user).execute();
}
```

```
Logic SQL: insert into user (id, name, phone, sex)
values (?, ?, ?, ?)

Actual SQL:insert into user_1 (id, name, name_ext, name_fuzzy
, phone, phone_fuzzy, sex) values (?, ?, ?, ?, ?, ?, ?) :::
[1, qNqrqBfHofBIdlUzEtSbPA==, qNqrqBfHofBIdlUzEtSbPA==, 詑顬刧瑨,
qEmE7xRzW0d7EotlOAt6ww==, 04101454589, 男]
```

原SQL插入字段只有4个，我name配置了加密列、辅助列、模糊列，phone配置了模糊列。所以改写的SQL增加了3个字段，相应的value按照各自配置的算法进行加密后入库。

- Update示例

```
@PutMapping("/user/{id}")
@Transactional(rollbackFor = Exception.class)
public long put(@PathVariable Long id, @RequestBody User user) {
    return sqlQueryFactoryAlter.update(qUser).populate(user)
            .where(qUser.id.eq(id)
                    .and(qUser.name.like( str: "%熊大%"))
            ).execute();
}
```

```
Logic SQL: update user
set name = ?, phone = ?, sex = ?

Actual SQL: update user_1
set name = ?, name_ext = ?, name_fuzzy = ?,
phone = ?, phone_fuzzy = ?, sex = ?
where user_1.id = ? and user_1.name_fuzzy like ?
::: [qyPX2iE4jasGZIUCJvP8Jg==, qyPX2iE4jasGZIUCJvP8Jg==
, 詑顬刧瑨yy, V0H95UPxrPKdTj4D5Zek+g==, 041111111111, 女,
1, %詑顬%]
```

修改字段同样增加3个，相应的value按照各自配置的算法进行加密后入库。模糊查询列改写到模糊列，值也保留%后加密。

集成使用示例— ShardingSphere-JDBC使用示例

ShardingSphere-JDBC使用示例

- 简单Select示例

```
@GetMapping("/user/{id}")
public User getOne(@PathVariable Long id) {
    return sqlQueryFactoryAlter.select(qUser).from(qUser)
            .where(qUser.id.eq(id)
            .and(qUser.name.like( str: "熊大%"))).fetchOne();
}
```

```
Logic SQL: select user.id, user.name, user.name_fuzzy, user.phone,
 user.phone_fuzzy, user.sex
from user user
where user.id = ? and user.name like ?
limit ?

Actual SQL: ds ::: select user.id, user.name AS name, user.name_fuzzy,
 user.phone AS phone, user.phone_fuzzy, user.sex
from user_1 user
where user.id = ? and user.name_fuzzy like ?
limit ? ::: [1, 侉在%, 2]
```

```
"id": 1,
"name": "熊大xxx",
"sex": "男",
"phone": "130000000000",
"phoneFuzzy": "041111111111",
"nameFuzzy": "侉在999"
```

- 多表查询示例

```
public Map multiTable(@PathVariable Long id) {
    return sqlQueryFactoryAlter
            .select(Projections.map(
                    qUser.id,
                    qUser.name,
                    qUser.sex,
                    qUserExt.address))
            .from(qUser).leftJoin(qUserExt)
            .on(qUserExt.id.eq(qUser.id))
            .where(qUser.id.eq(id)).and(qUser.phone.like( str: "130%")))
            .fetchOne();
```

```
Logic SQL: select user.id, user.name, user.sex,
user_ext.address from user user
left join user_ext user_ext
on user_ext.id = user.id
where user.id = ? and user.phone like ?
limit ?

Actual SQL: ds ::: select user.id, user.name AS name, user.sex,
user_ext.address from user_1 user
left join user_ext user_ext
on user_ext.id = user.id
where user.id = ? and user.phone_fuzzy like ?
limit ? ::: [1, 041%, 2]
```

```
"user.name": "熊大xxx",
"user.sex": "男",
"user_ext.address": "武汉",
"user.id": 1
```

集成使用示例— ShardingSphere-JDBC使用限制

ShardingSphere-JDBC使用限制

- 数据加密模块限制

  加密字段无法支持查询不区分大小写功能；

  加密字段无法支持比较操作，如：大于、小于、ORDER BY、BETWEEN、LIKE 等；

  加密字段无法支持计算操作，如：AVG、SUM 以及计算表达式。

- 数据分片模块限制

  无法支持：

  - CASE WHEN 中包含子查询
  - CASE WHEN 中使用逻辑表名（请使用表别名）

  不稳定支持：

  - 子查询和外层查询未同时指定分片键，或分片键的值不一致时。
  - 当关联查询中的多个表分布在不同的数据库实例上时

我们测试了一下分片、加密一起使用的场景，可以看到常见SQL场景基本支持。

```
String sql = "insert into user ( id, name ) values ( 1, '熊大' ) on duplicate key update name = '324546565'";
//单表
sql = "insert into user ( id, name,sex ) values ( 1, '熊大','男' )";
sql = "insert into user (id, name,sex) values ( 3, '熊大','男'), ( 4, '熊大','男')";

sql = "delete from user where sex = '男' and name like '熊%'";

sql = "update user set name = '熊大123', sex = '男' where sex ='男' and phone like '130%'";

sql = "select * from user";
sql = "select id, name, phone, sex from user where id = 1 and name like '熊%'";
sql = "select * from user where (id = 1 or phone = '13012345678') and name like '熊%'";
sql = "select * from user where name like '熊%' order by name desc limit 1 ";
sql = "select count(id) from user where  name like '熊%' group by sex order by sex desc limit 1";
sql = "select distinct * from user where name like '熊%'";
sql = "select distinct sex from user where name like '熊%'";

//不支持
sql = "insert into user_0 values (2, '熊大','男')";
sql = "insert into user (id, name) values(1+2,'xgx')";
sql = "insert into user (id, name) select id ,name from user where id =1";
sql = "select count(id) from user where  name like '熊%' group by name order by name desc limit 1";
sql = "select distinct name from user where name like '熊%'";

//子查询
sql = "select * from user where id in (select id from user where phone = '130' and name like '熊大%' )";
//不支持
sql = "select user.* from (select * from user_0 where id in (select id from user_0 where" +
      " phone = '130' and name like '熊%' )) as user";
//连表
sql = "select * from user LEFT JOIN user_ext on user.id=user_ext.id where user.sex='男' and user.name like '熊%'";
sql = "select * from user LEFT JOIN user_ext on user.id=user_ext.id where user.id in (select id from " +
      "user where sex = '男' and name like '熊%')";
sql = "select user.id from user  where name like'熊%' UNION select user_ext.id from user_ext ";

//不支持
sql = "UPDATE user SET name = CASE WHEN id = 1 THEN 'nj' WHEN id = 2 THEN 'wh' END,sex = CASE WHEN " +
      "id = 1 THEN 'abc' WHEN id = 2 THEN 'def' END WHERE id IN (1,2)";
```

集成使用示例一 ShardingSphere-JDBC多数据源

ShardingSphere-JDBC多数据

- SQL不支持、不能加密。

  有的SQL无法执行，无法自动加解密怎么办。

我们可以用原生数据源自己加解密去操作业务，

下面介绍下Querydsl和Mybatis下多数据源方案。

- 配置数据源

```
spring:
  datasource:
    mysql:
      username: root
      password: root
      jdbcUrl: jdbc:mysql://127.0.0.1:3306/test
      driver-class-name: com.mysql.cj.jdbc.Driver
```

```
@Bean(name = "mysqlDatasource")
@ConfigurationProperties(prefix = "spring.datasource.mysql")
public DataSource mysqlDatasource() { return DataSourceBuilder.create().build(); }
```

- Querydsl

```
@Bean(name = "queryFactoryAlter")
public SQLQueryFactory queryFactoryAlter(@Qualifier("shardingSphereDataSource") DataSource dataSource)
    com.querydsl.sql.Configuration configuration = new com.querydsl.sql.Configuration(MySQLTemplates.bu
    configuration.addListener(new SpringSqlCloseListener(dataSource));
    return new SQLQueryFactoryAlter(
            configuration,
            () -> DataSourceUtils.getConnection(dataSource));
}

@Bean(name = "queryFactory")
public SQLQueryFactory queryFactory(@Qualifier("mysqlDatasource") DataSource dataSource) {
    com.querydsl.sql.Configuration configuration = new com.querydsl.sql.Configuration(MySQLTemplates.bu
    configuration.addListener(new SpringSqlCloseListener(dataSource));
    return new SQLQueryFactoryAlter(
            configuration,
            () -> DataSourceUtils.getConnection(dataSource));
}
@Autowired
@Qualifier("queryFactoryAlter")
private SQLQueryFactory sqlQueryFactoryAlter;

@Autowired
@Qualifier("queryFactory")
private SQLQueryFactory sqlQueryFactory;
```

1、配置好DataSource

2、选择不同的数据源，生成两个SQLQueryFactory。

3、使用时选择不同SQLQueryFactory注入

集成使用示例— ShardingSphere-JDBC多数据源

ShardingSphere-JDBC多数据

- Mybatis。

```
@Bean(name = "mysqlSessionFactory")
public SqlSessionFactory mysqlSessionFactory(
        @Qualifier("mysqlDatasource") DataSource dataSource)
        throws Exception {
    SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
    bean.setDataSource(dataSource);
    //设置XML文件存放位置
    bean.setMapperLocations(new PathMatchingResourcePatternResolver()
            .getResources( locationPattern: "mapper/original/*.xml"));
    return bean.getObject();
}

@Bean(name = "shardingSphereSessionFactory")
public SqlSessionFactory shardingSphereSessionFactory(
        @Qualifier("shardingSphereDataSource") DataSource dataSource)
        throws Exception {
    SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
    bean.setDataSource(dataSource);
    //设置XML文件存放位置
    bean.setMapperLocations(new PathMatchingResourcePatternResolver()
            .getResources( locationPattern: "mapper/shardingsphere/*.xml"));
    return bean.getObject();
}
```

```
@MapperScan(basePackages = "com.gxxiong.shardingsphere.mapper.original",
        sqlSessionFactoryRef = "mysqlSessionFactory")
@MapperScan(basePackages = "com.gxxiong.shardingsphere.mapper.shardingsphere",
        sqlSessionFactoryRef = "shardingSphereSessionFactory")
```

```
@Autowired
private UserOriginalMapper originalMapper;

@Autowired
private UserShardMapper userShardMapper;
```

1、配置好DataSource

2、原生SQL和Sharding SQL放在不同mapper下

3、选择不同的数据源，生成不同的SqlSessionFactory

4、启动的时候@MapperScan扫描两个数据源下的mapper

5、使用时选择不同mapper注入即可

欢迎大家合作共建